

Ускоряем хранимые процедуры
на Postgres pl/pgSQL
по гистограммам,
или Жизнь после импортозамещения
Анатолий Анфиногенов
HL
HighLoad++
2022

Краткое содержание предыдущих серий

2020 Как я перестал беспокоиться и перенес 60К строк из 150 процедур PL/SQL в Postgres
https://pgconf.ru/2020/274661

2021 Миграция приложения Oracle PL/SQL на Postgres pl/pgSQL: взгляд два года спустя
https://pgconf.ru/202110/308499

2021 Миграция приложения Oracle PL/SQL на Postgres pl/pgSQL: планирование, подготовка, переход и два года жизни с новой БД
https://conf.ontico.ru/lectures/3829870

2022 Жизнь после импортозамещения: некоторые особенности настройки БД и хранимых процедур
https://pgconf.ru/2022/316102



Что это и зачем это нужно?

- ЭЛЬБРУС (2006 г. – н.в.) – система планирования движения грузовых поездов по энергооптимальным расписаниям

- ЭЛЬБРУС работает на всех железных дорогах России от Калининграда до Хабаровска





Первая премия UIC/МСЖД–2012 в области железнодорожных исследований и инноваций



Так выглядит расписание поездов снаружи

Проект Направления Вид Загрузка данных Проверка графика Построение графика Аналитика Энергооптимизация Экспорт данных Редактирование графика Базовый график Сквозной график Стыковка графиков Инструменты Помощь Пакеты

Подход Прогноз "Окна" Построение

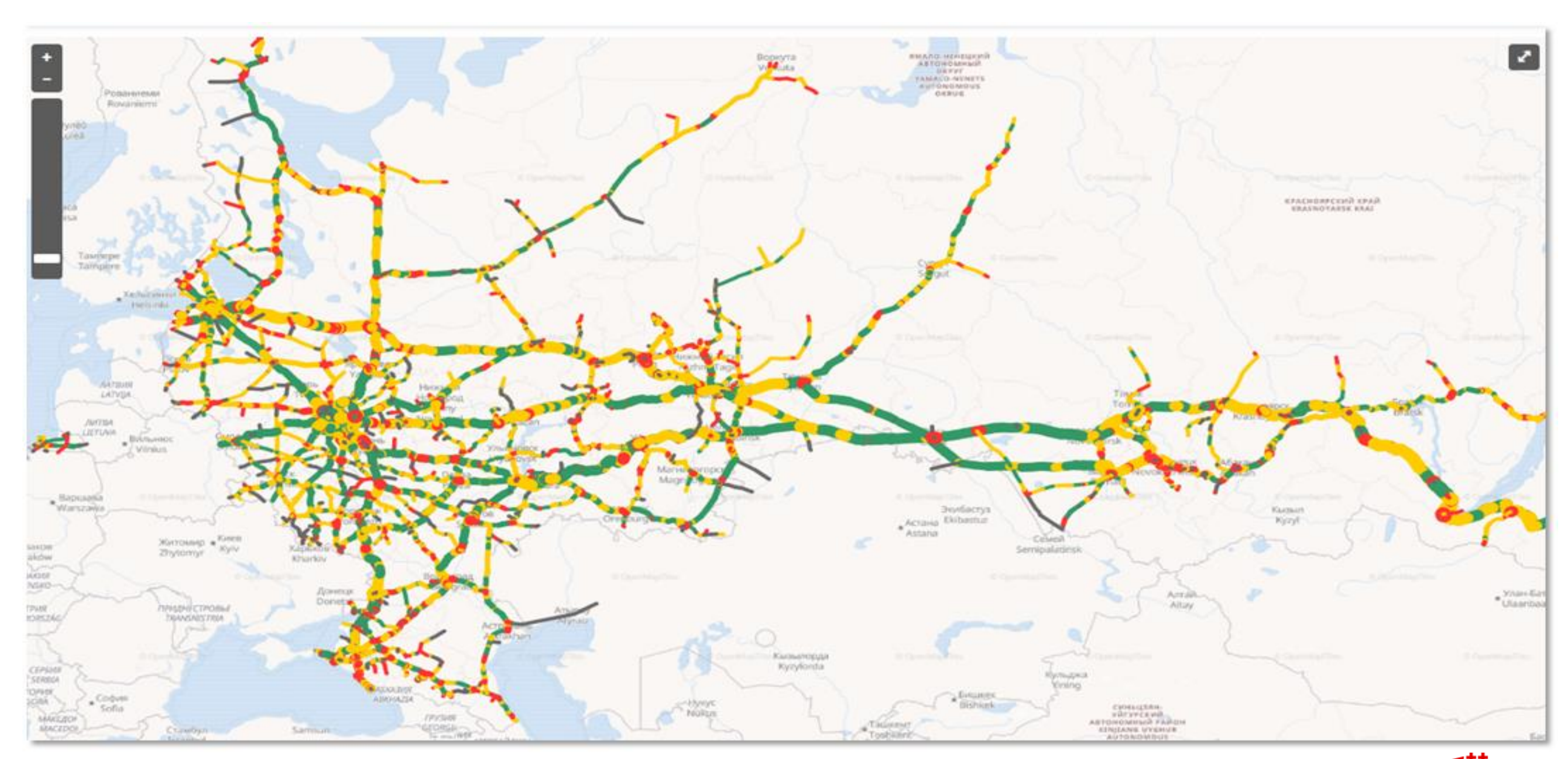
Исилькуль
Кара-Гуга
Булаево_I
Ганькино
Токуши
Петропавловск
Кондратовка-Сибирская
Мамлютка
Горбуново
Петухово
Пьянково
Макушино
Коновалово
О.п 2459
Лебяжья-Сибирская
Кравцево
О.п 2417
Варгаши
Утяк
Курган
Введенское
Зырянка
Юргамыш
Сладкое
Мишкино
Бутырское
Шумиха
О.п 2210
Щучье
Алакуль
Каясан
Чернявская
Козырево
Чурилово
Челябинск_В

09.04.2014
10.04.2014
11.04.2014
12.04.2014

область архивного графика
область действующего графика
область прогнозного графика



Так выглядит прогнозная аналитика



HL HighLoad++ 2022

Так устроено расписание поездов изнутри

- 1 расписание = до 2 млн объектов 20 типов
- 1 расписание = до 500 Мбайт для 1 железной дороги
- 16 железных дорог России += ежедневно
  10-30 расписаний разных типов
- 1 дорога = 500 расписаний в оперативной базе
- Центральная архивная БД = история расписаний
  и прогнозная аналитика



Архитектура ЭЛЬБРУС: трехзвенная

- Распределенность: 16 узлов на железных дорогах и сервера центрального уровня
- Эксплуатация: 24/7; регулярные обновления серверных приложений 3-6 раз в год, включая приложения БД
- Толстый клиент: Windows, C++, очереди (ActiveMQ)
- Тонкий клиент: GWT, Angular
- Сервер приложений: Java, Tomcat, очереди (ActiveMQ)
- Сервер БД: **ванильный Postgres 11/13**



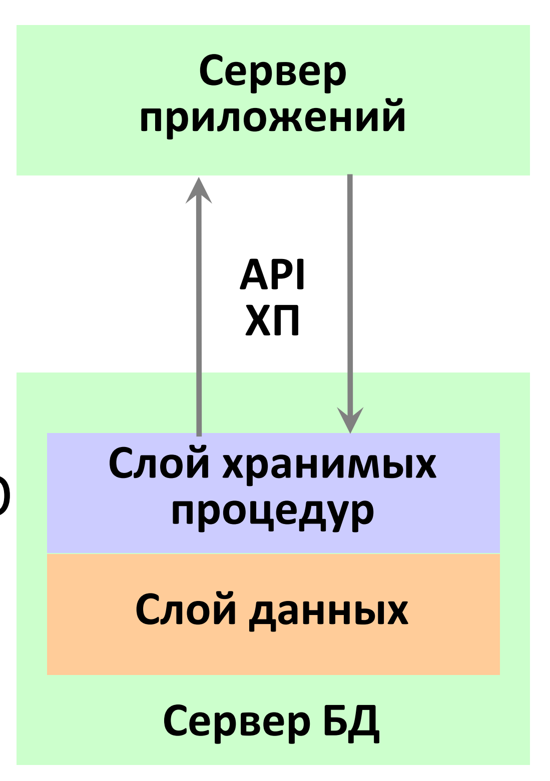
Особенности базы данных

- ~250 таблиц, 250 Гб оперативных данных, 2 Тб архивных
- Обновление до 40-60% содержимого БД за неделю
- ~200 хранимых процедур на `pl/pgSQL` (~50000 строк) в отдельной схеме
- Эмуляция (`dblink` + `pg_variables`) автономных транзакций для логирования вызовов API
- Временные таблицы в стиле Oracle для обмена данными с сервером приложений (сотни Мбайт) при вызовах хранимых процедур
- 10-50 одновременно работающих клиентов; очень крупные, но относительно редкие транзакции
- Работа с БД – только API хранимых процедур



Почему только хранимые процедуры?

- Взаимодействие в предметных категориях приложения
- Логика хранения с помощью API ХП отделена от бизнес-логики приложения
- Можно вносить изменения в структуры данных и организацию БД без изменения сервера приложений (в пределах API)
- Возможно гладкое поэтапное обновление распределенного ПО счет версионности API
- Встроенный механизм диагностики, логирования, отладки и профилирования приложения БД без остановки сервиса, управляемый параметрически





Что нового?

1) Новые приключения

- Опыт эксплуатации БД приложения для PostgreSQL: **переход от борьбы с детскими болезнями к непрерывной диагностике и лечению хронических заболеваний**

- Опыты в части перехода от свободного ПО к отечественному ПО: CentOS –> RedOS/AstraLinux/?…

2) Разработка продолжается: ЭЛЬБРУС-М

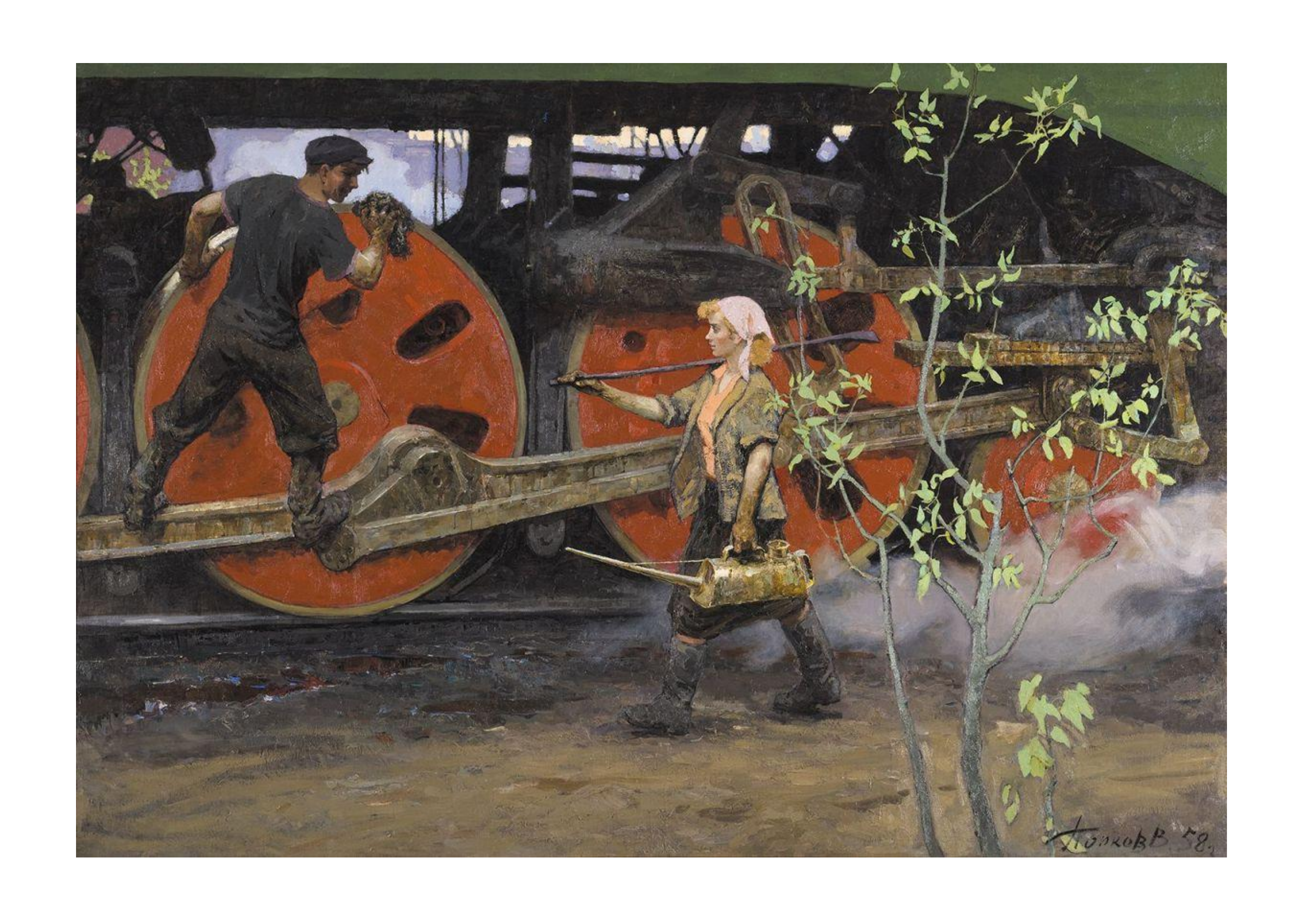


- ЭЛЬБРУС–М (М – макромодель) для прогноза продвижения поездопотоков и оценки инфраструктурных и управляющих решений

- Проектируем и реализуем ЭЛЬБРУС–М правильно с учетом особенностей PostgreSQL и опыта ЭЛЬБРУС!



Импортозамещаемся



Импортозамещение в БД ЭЛЬБРУС

Разработка

Отработаны инфраструктурные решения, созданы структуры данных и перенесены 200 процедур (60000 строк) на PostgreSQL 11

Эксплуатация: детские болезни

Создание индексов, работа с временными таблицами, оптимизация SQL-команд



Старт перехода

Пилотный проект, к началу 2019 г. перенесено 11 процедур на PostgreSQL 10

Переход

Параллельная эксплуатация старой и новой БД в переходный период, перенос данных, начальная настройка производительности

Эксплуатация: хронические заболевания

Настройка сбора статистики и автовакуума; борьба с «распуханием» таблиц и индексов



Проблема выбора: как переносить БД+ХП?



? Переносим как есть

? Перенос и мини-рефакторинг

? Напишем заново как следует





Пошаговая история импортозамещения – 1

1. Разработка общей технологии миграции: зачем, что, когда и как следует сделать.
2. Обучение особенностям работы с PostgreSQL.
3. Разработка инфраструктурных workaround, без которых наше приложение не перенести.
4. Перенос таблиц, VIEW, SEQUENCES и т. п.
5. Перенос (возможно, с переделкой) хранимых процедур.
6. Тестирование, отладка и первоначальная оптимизация производительности приложения БД.



Пошаговая история импортозамещения – 2

7. Разработка технологического процесса переключения.
8. Разработка эксплуатационной документации и инструкций для администраторов.
9. Обучение администраторов.
10. Переключение и начало переходного периода параллельной эксплуатации.
11. Организация сопровождения приложения и БД, включая мониторинг.
12. Разработка и внедрение технологии выполнения обновлений.



Пошаговая история импортозамещения – 3

13. Выявление и преодоление «детских болезней».
14. Изучение основ управления производительностью PostgreSQL и применение их на практике.
15. Пересмотр и расширение номенклатуры метрик мониторинга приложения и БД.
16. Расширение перечня расширений (sic!).
17. Диагностика «хронических заболеваний» и их лечение.

← Вы находитесь здесь

18. Переход с ванильной PostgreSQL на PostgresPro.
19. ???



Изобретаем обходные решения



Обходные инфраструктурные решения – 1

- Хранимые процедуры пишут логи в лог-таблицы вне зависимости от успешности транзакции – **нужны автономные транзакции**
- В ванильном PostgreSQL автономных транзакций нет
- Их эмуляция через **`dblink`** работает медленно, т.к. создание соединения – дорогая операция
- **Спасает** расширение **`pg_variables`; спасибо, Иван Фролков!** В переменной храним открытое соединение, и логи начинают писаться с приемлемой скоростью
- Если нельзя использовать расширение **`pg_variables`**, его могут заменить **custom variables** в **`postgresql.conf`**
(см. custom_variable_classes)



Обходные инфраструктурные решения – 2

- Для приема больших объемов данных от сервера приложений при вызове хранимых процедур **нужны временные таблицы (в стиле Oracle!)**
- Ванильный PostgreSQL не поддерживает временные таблицы, сохраняющиеся в словаре данных СУБД по завершении сессии
- **Неверное решение:** их эмуляция с помощью UNLOGGED-таблиц с уникальным GUID слоя данных
- **Верное решение:** изменение API путем обязательного вызова функции `prepareCall('`*ИМЯ_ПРОЦЕДУРЫ*`')` для создания временных таблиц в данной сессии

Об UNLOGGED-таблицах в нескольких словах

as you value your life or your reason keep away from

…UNLOGGED TABLES



Обходные инфраструктурные решения – 3

- Для выполнения задач архивирования устаревших данных, техобслуживания БД, диагностики и проактивного мониторинга мы использовали **пакетные задания (JOB) Oracle**, которые вызывали процедуры **с оператором COMMIT внутри**
- Ванильный PostgreSQL не поддерживает JOB
- Для запуска процедур, реализующих эти функции, было создано Java-приложение **`jobrunner`** для Apache Tomcat. Плюсы: оно гибче, частично унифицировано с сервером приложений и может выполнять не только задания БД. Минусы: более сложная конструкция

Запомнившиеся моменты



- В Oracle тип DATE подобен TIMESTAMP с меньшей точностью

  TO_DATE('2020-10-15 14:00','YYYY-MM-DD HH24:MI') = 2020-10-15 14:00

  В PostgreSQL тип DATE округляется до даты

  TO_DATE('2020-10-15 14:00','YYYY-MM-DD HH24:MI') = 2020-10-15

- При переносе операторов MERGE из Oracle они заменялись на INSERT(…) ON CONFLICT DO UPDATE…, но в случае отсутствия основания для конфликта (т.е. первичного ключа или уникального индекса) ветвь DO UPDATE… не выполнялась никогда, не вызывая замечаний с точки зрения синтаксиса

(В PostgreSQL 15 появился оператор MERGE)



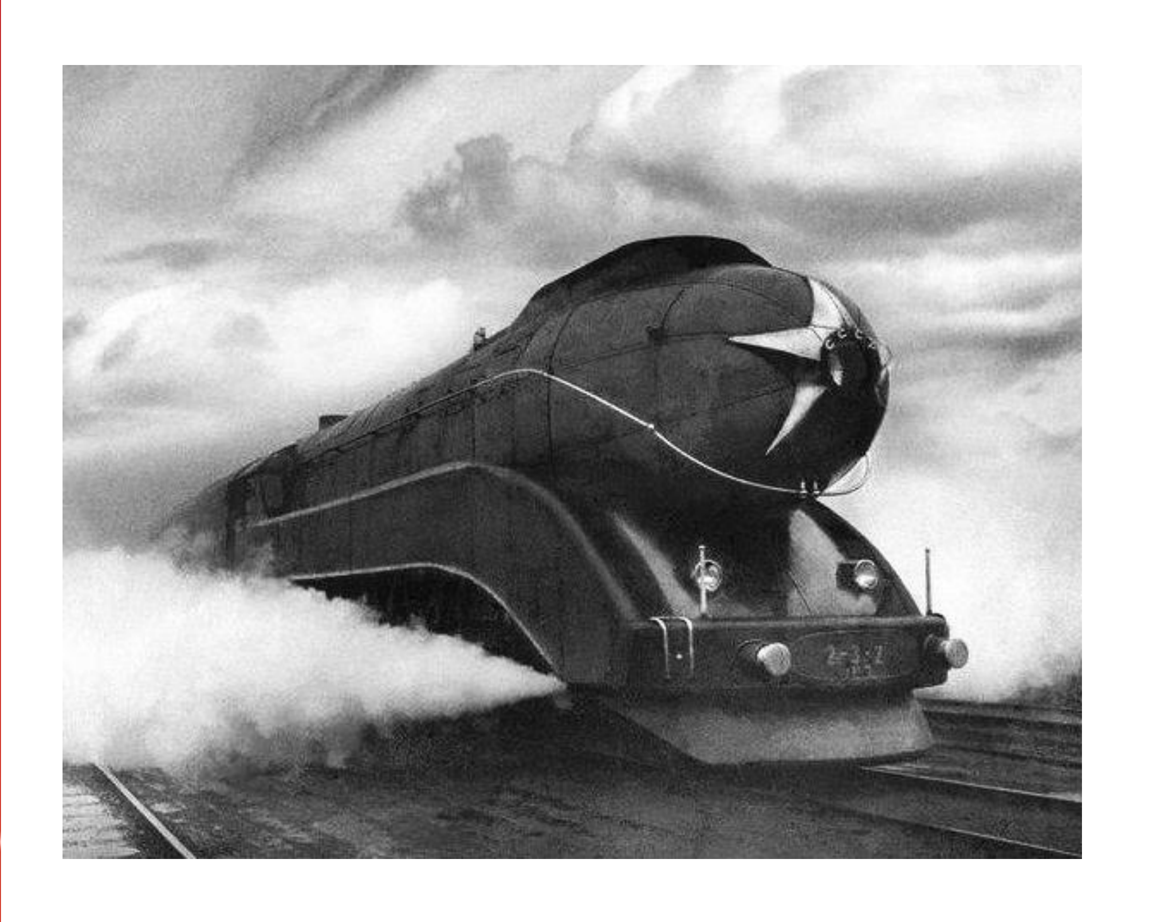
Используемые расширения

Расширение	Назначение в проекте	Штатное?
dblink	Эмуляция автономных транзакций	да
pg_variables	Эмуляция автономных транзакций	**нет** (да в Pro)
pgcrypto	Генерация GUID	да
plpgsql_check	Проверка хранимых процедур	**нет**
plpythonu	Работа с файлами в ФС сервера	да
postgres_fdw	Связь с архивными серверами	да
oracle_fdw	Интеграция со смежными системами	**нет** (да в Pro)
pg_stat_statements	Мониторинг производительности БД	да
pgstattuple	Проверка «распухания» таблиц и индексов	да
pg_stat_st	Лечение «распухания» таблиц и индексов	да

Нештатные расширения затрудняют обновление СУБД!



Ускоряемся по гистограммам



Где измерять производительность ХП?

Место измерения	+	−
На стороне сервера приложений	Низкие накладные расходы	Нет подробностей: что, где, почему и как это исправить
Изнутри хранимой процедуры	Максимальная точность и информативность. Данные о скорости работы ХП внутри БД	Накладные расходы. Необходимость модификации кода
Со стороны БД	Не надо модифицировать приложение	Недостаточная информативность. Накладные расходы



Профилирование хранимых процедур

- Возможность профилировать хранимые процедуры должна быть заложена **при проектировании**
- Процедура разбита на **этапы** (один или несколько сходных по типу SQL-операторов); времена их выполнения сохраняются в логи производительности
- Профилирование должно включаться и управляться параметрически с возможностью отключения; времена выполнения должны попадать в **мониторинг**
- Необходимо **отслеживать планы выполнения проблемных запросов изнутри хранимой процедуры** (было в ХП для Oracle, но это ~~невозможно~~ сложно в ванильной PostgreSQL — печаль)



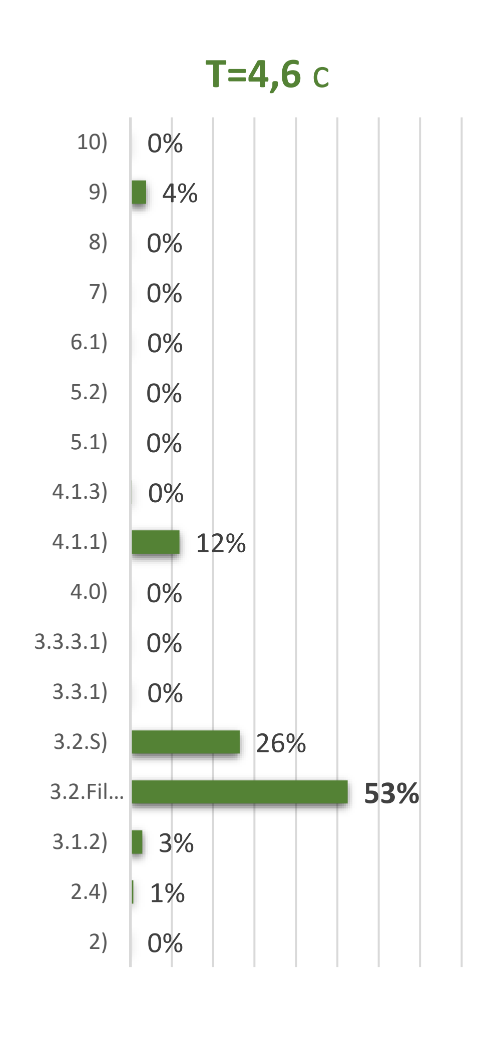
Этапы хранимой процедуры: пример

2)	Чтение и проверка входных параметров
2.4)	Создание рабочих временных таблиц
3.1.2)	INSERT,SELECT (4K) : Чтение во временную таблицу всех поездов расписания с первичной фильтрацией
3.2.Filter)	INSERT,SELECT (1112K) : Чтение во временную таблицу всех точек расписания с первичной фильтрацией
3.2.S)	INSERT,SELECT (4K) : Чтение во временную таблицу лишних поездов по доп.критериям фильтрации
3.3.1)	DELETE: Удаляем лишние поезда из таблицы лишних поездов
3.3.3.1)	DELETE: Удаляем поезда с признаком неактивности
4.0)	ANALYZE: Анализируем временную таблицу поездов
4.1.1)	INSERT,SELECT (1000K) : Чтение во временную выходную таблицу точек расписания
4.1.3)	ANALYZE: Анализируем временную выходную таблицу точек расписания
5.1)	DELETE: Дополнительная тонкая фильтрация точек расписания
5.2)	DELETE: Дополнительная тонкая фильтрация точек расписания
6.1)	UPDATE: Вычисление и заполнение серийных номеров точек расписания при необходимости
7)	DELETE: Дополнительная фильтрация расписания по полигонам
8)	DELETE: Удаляем поезда без точек или удовлетворяющие доп.критериям
9)	INSERT,SELECT (225K) : Чтение во временную таблицу всех календарей отобранных поездов расписания
10)	SELECT: Открываем курсоры к временным таблицам и возвращаем результаты

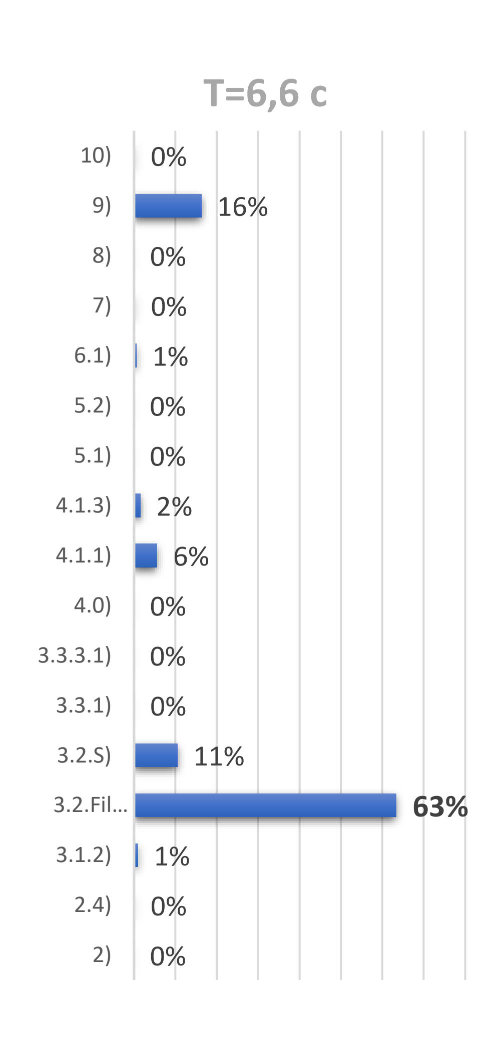
`getTrainsData()`: чтение расписания со сложной фильтрацией



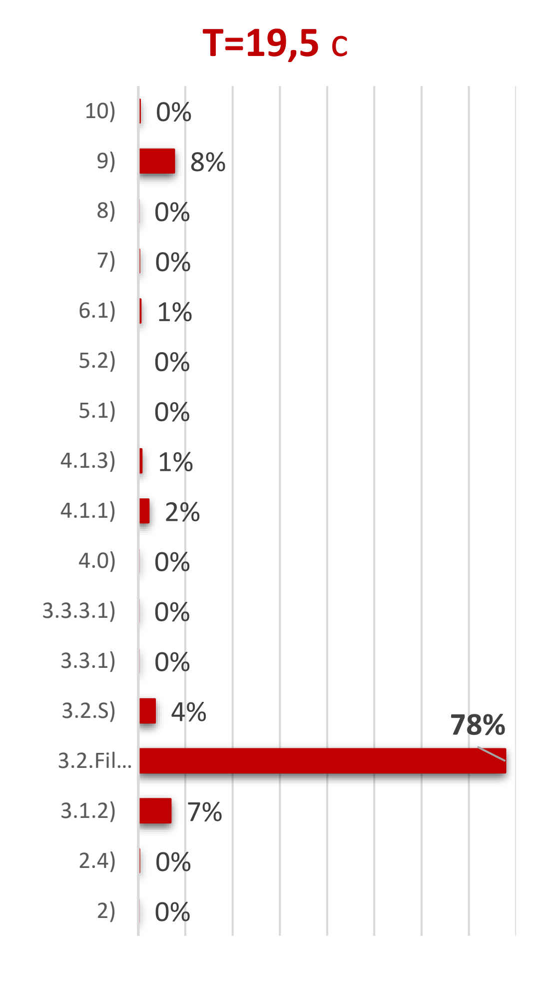
Гистограммы: THE GOOD, THE BAD AND THE UGLY



отлично



приемлемо



плохо



Ускоряем процедуру по гистограмме

Ускорение, раз	Этап процедуры	T, мс (до/после ускорения)
3,00	2)	6
	2)	2
1,30	2.4)	13
	2.4)	10
1,40	3.1.2)	2309
	3.1.2)	1648
17,09	3.2.S)	48468
	3.2.S)	2836
1,04	3.3.1)	55
	3.3.1)	53
0,89	3.3.3.1)	24
	3.3.3.1)	27
	4.0)	20
497,54	4.1)	97517
	4.1.1)	67
	4.1.3)	82
200,00	5.1)	2
	5.1)	0
	5.2)	0
	5.2)	0
1,00	6.1)	6
	6.1)	6
0,83	7)	10
	7)	12
1,00	8)	18
	8)	18
1,55	9)	22812
	9)	14733
1,38	10)	11
	10)	8

- Это — пример итерационного подхода к ускорению процедуры `getTrainsData()`
- Пик на гистограмме — повод разобраться с SQL-операторами проблемного этапа
- Эффективность приемов оптимизации оцениваем по времени выполнения этапов
- В этом примере процедура ускорилась в 9 раз: было 171 с, стало 19 с.
- Основной эффект видим на этапах «4.1)», «3.2.S)» и «9)», см. знак
- Знаки — цели дальнейшей оптимизации



Гистограмма до и после оптимизации

Распределение относительного времени выполнения в %
по этапам процедуры getTrainsData

Доля времени этапа, %

90%
80%
70%
60%
50%
40%
30%
20%
10%
0%

до оптимизаци
после оптимизаии

Снижаем главный пик

Доля других этапов растёт

2) 2.4) 3.1.2) 3.2.Filter.Other.3) 3.2.S) 3.3.1) 3.3.3.1) 4.0) 4.1.1) 4.1.3) 5.1) 5.2) 6.1) 7) 8) 9) 10)

Этап выполнения процедуры

- Гистограмма времени выполнения этапов процедуры позволяет выявить **«узкие места» — этапы, на которые ушло более чем 80% времени выполнения процедуры**
- **Цель — по возможности более равномерная гистограмма!**

Как это было достигнуто?



Лечим детские болезни



HighLoad++ 2022

«Детские болезни» производительности – 1

Лежащие на поверхности способы ускорения:

1. В отличие от Oracle, в PostgreSQL внутри хранимых процедур можно не только создавать таблицы, но и собирать статистику. (Выясните, а почему ранее это не сделал автовакуум?!)

   Решение: ANALYZE TMP_InputData;

   Результат: **ускорение в 1,5 раза**

2. Отсутствие индекса на таблице, которая выросла в размерах. Основная сложность – осознать это.

   Решение: CREATE INDEX …;

   Результат: **ускорение в 14000 раз**



«Детские болезни» производительности – 2

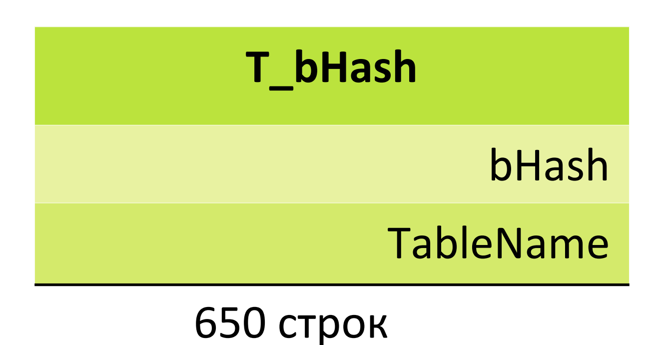
Проблема *«что-то база тормозит»*

```
SELECT query, calls, total_exec_time, mean_exec_time
FROM   pg_stat_statements
ORDER BY total_exec_time DESC LIMIT 10;
```

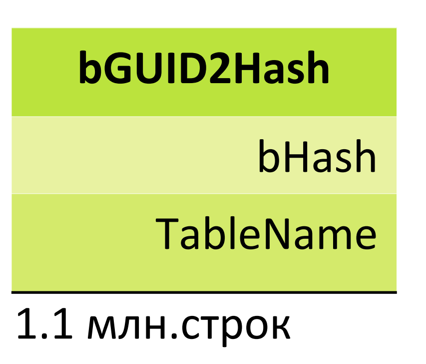
query	calls	total_exec_time	mean_exec_time
select * from M.getHashInfo($1,$2,$3,$4) as	40,827	124,643,313.050181	3,052.96282
FETCH ALL IN "P_RESULTSET"	159,883	41,421,988.167746	259.076876
select * from	13,476	33,627,829.13626	2,495.386549
SELECT $2 FROM ONLY "m"."tt_train_links" x	43,695	30,163,201.28449	690.312422
select * from	13,418	13,552,919.840955	1,010.055138
SELECT	13,418	13,157,557.284014	980.590049
select * from M.saveT_Threads($1,$2,$3) as	12,219	3,569,138.341333	292.097417
select * from M.saveT_Trains($1,$2,$3) as	13,083	3,527,987.256952	269.661947
select * from EAPI.doMonitoringJob() as	73,761	1,644,611.695368	22.296494
SELECT dblink_exec(v_ConnName,v_SQL)	837,924	1,489,619.495458	1.77775

Суть проблемы «что-то база тормозит»

Разберемся с первым запросом из Топ-10 pg_stat_statements:



650 строк



1.1 млн.строк

bGUID2Hash имеет индекс (PK) по bHash

```
SELECT H.*
FROM   T_bHash T
       INNER JOIN bGUID2Hash H
       ON(   (T.bHash=H.bHash)
         AND(T.TableName=H.TableName)
        )
```

QUERY PLAN
Hash Left Join (cost=929678.45..1137426.73 rows=650 width=96)
  Hash Cond: ((t.bhash = h.bhash) AND (t.table_name = h.tablename))
  -> Seq Scan on t_bhash t (cost=0.00..16.50 rows=650 width=96)
  -> Hash (cost=432011.98..432011.98 rows=19332698 width=61)
        -> **Seq Scan** on bguid2hash h (cost=0.00..432011.98 rows=19332698 width=61)"Planning Time: 5.428 ms
Planning Time: 896.667 ms
Execution Time: 9532.021 ms



Решение проблемы «что-то база тормозит»

Детская ошибка: таблица bGUID2Hash подросла
и Seq Scan по ней стал узким местом – необходим индекс

```
CREATE UNIQUE INDEX bGUID2Hash_IDX1 ON bGUID2Hash(bHash, TableName)
```

```
QUERY PLAN
Nested Loop Left Join  (cost=0.56..5592.75 rows=650 width=96) (actual time=0.292..0.522 rows=3 loops=1)
  ->  Seq Scan on t_bhash t  (cost=0.00..16.50 rows=650 width=96)
      (actual time=0.020..0.025 rows=3 loops=1)
  ->  Index Scan using bguid2hash_idx1 on bguid2hash h  (cost=0.56..8.58 rows=1 width=61)
      (actual time=0.153..0.153 rows=1 loops=3)
        Index Cond: ((bhash = t.bhash) AND (tablename = t.table_name))
Planning Time: 2.212 ms
Execution Time: 0.682 ms
```

Ускорение в 14000 раз!



А сколько стоит логирование?

Суммируем времена запросов логирования из pg_stat_statements и сравниваем их с общим временем выполнения запросов:

query	calls	total_exec_time	mean_exec_time
select * from M.getHashInfo($1,$2,$3,$4) as	40,827	124,643,313.050181	3,052.96282
FETCH ALL IN "P_RESULTSET"	159,883	41,421,988.167746	259.076876
select * from	13,476	33,627,829.13626	2,495.386549
SELECT $2 FROM ONLY "m"."tt_train_links" x	43,695	30,163,201.28449	690.312422
select * from	13,418	13,552,919.840955	1,010.055138
SELECT	13,418	13,157,557.284014	980.590049
select * from M.saveT_Threads($1,$2,$3) as	12,219	3,569,138.341333	292.097417
select * from M.saveT_Trains($1,$2,$3) as	13,083	3,527,987.256952	269.661947
select * from EAPI.doMonitoringJob() as	73,761	1,644,611.695368	22.296494
SELECT dblink_exec(v_ConnName,v_SQL)	837,924	1,489,619.495458	1.77775

Общее время, мс	Логирование, мс	Потери, %
307,756,480	2,013,985	0.65

Логирование можно не отключать никогда!
(в нашем случае)



«Детские болезни» производительности – 3

3. Использование специфических только для PostgreSQL нестандартных форм операторов UPDATE и DELETE часто в нашем коде **дает прирост скорости до 40%** по сравнению с использованием подзапросов вида:

... WHERE EXISTS (SELECT ...) или ... WHERE X IN (SELECT ...)

<u>Примеры:</u>

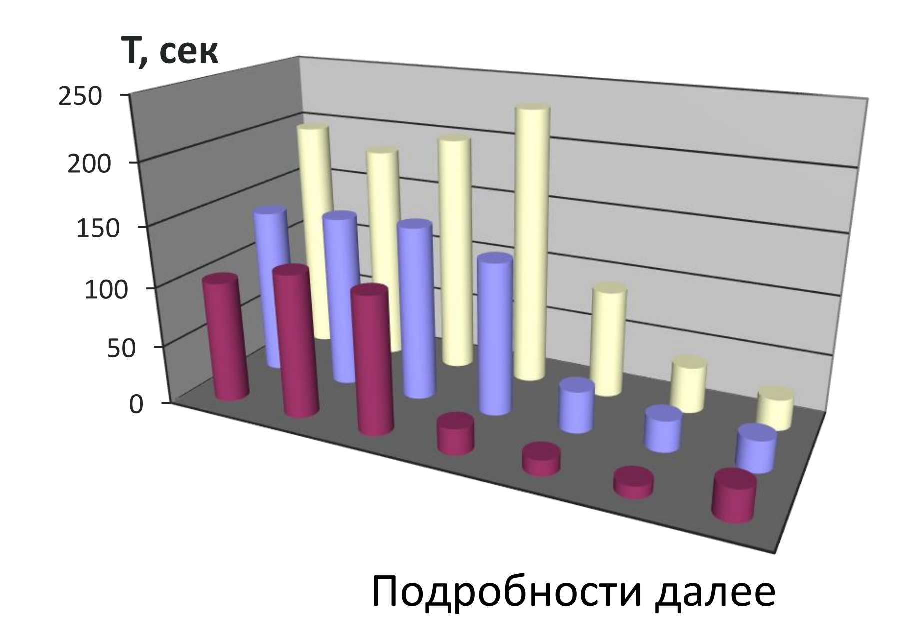
```
UPDATE IDs
SET    ID = Q.NEW_ID
FROM   (SELECT NEW_ID, ID FROM TMP_IDs) AS Q
WHERE (Q.ID = IDs.ID);

DELETE FROM IDs
USING  TMP_IDs
WHERE (IDs.ID=TMP_IDs.ID);
```



«Детские болезни» производительности – 4

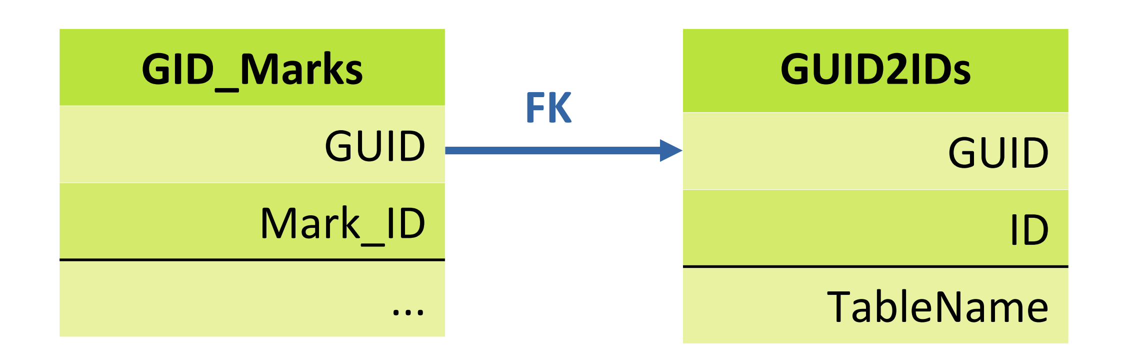
4. Операции удаления постепенно стали выполняться слишком медленно — проблема с ON DELETE CASCADE



Подробности далее

После принятых мер скорость удаления устаревших расписаний **возросла в среднем в 5 раз**

Суть проблемы с ON DELETE CASCADE



FOREIGN KEY (GUID)
REFERENCES GUID2ID(GUID)
ON DELETE CASCADE

Ванильный PostgreSQL 11.11

При выполнении оператора

DELETE FROM GUID2ID … ,

который должен был удалить
5 млн строк из **GUID2ID** и заодно из
GID_Marks,
оказалось, что БД выполнила **5 млн отдельных запросов** вида: Sec Scan

```
DELETE FROM GID_Marks
WHERE GUID = '4e423f41'
```



Решение проблемы с ON DELETE CASCADE

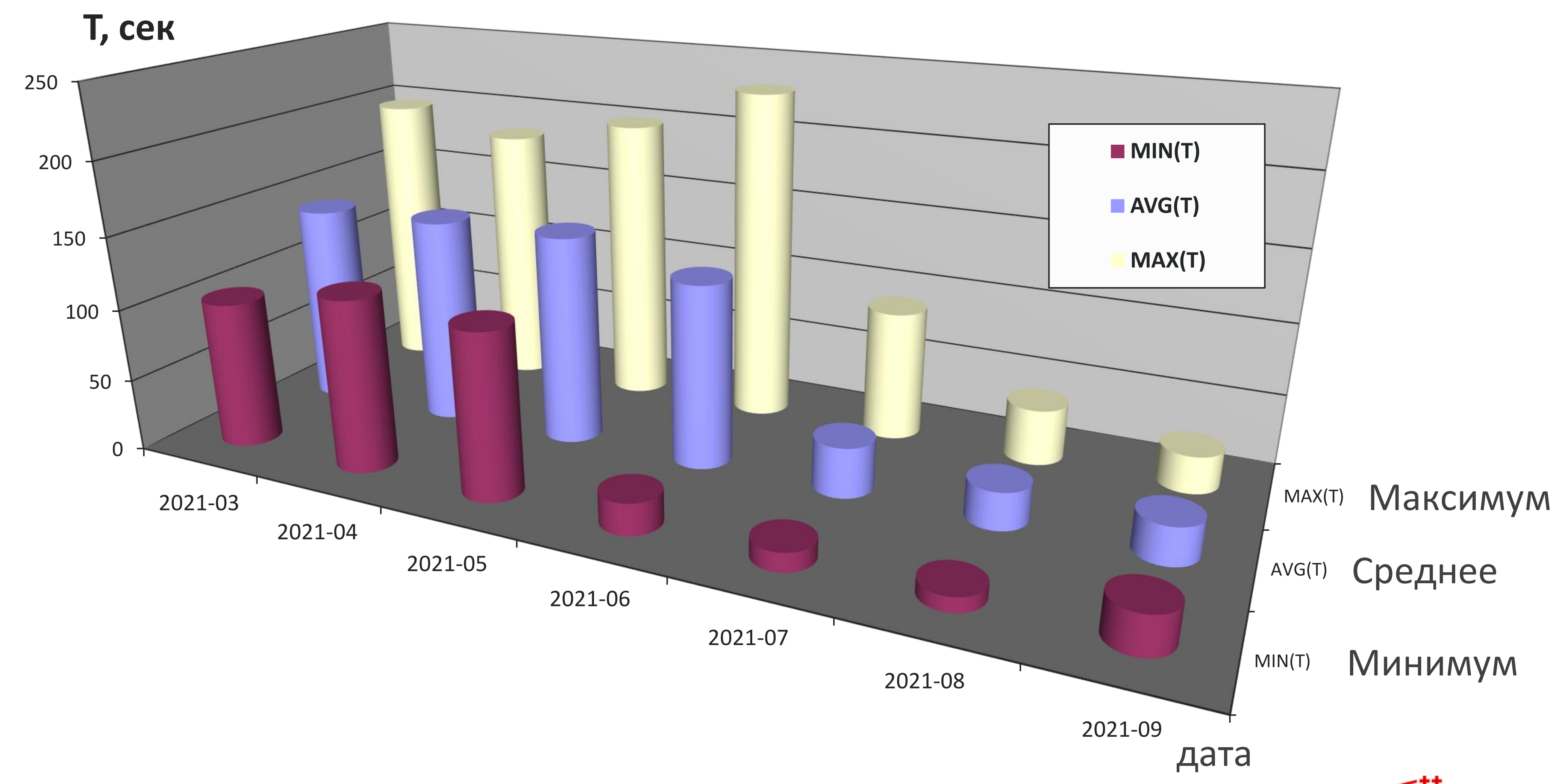
1) Разорвать связь ON DELETE CASCADE

2) Для эффективного одновременного удаления из родительской и дочерней таблицы использовать
SQL-операторы следующего вида:

```
WITH CTE_GUIDs AS
   (
     DELETE FROM GID_Marks
     WHERE (GID_Marks.TimetableID = v_TimetableID)
     RETURNING GID_Marks.GUID
   )
DELETE FROM GUID2ID
USING  CTE_GUIDs
WHERE (GUID2ID.GUID=CTE_GUIDs.GUID);
```



Время выполнения deleteTimetable



Проблема с удалением расписаний решена

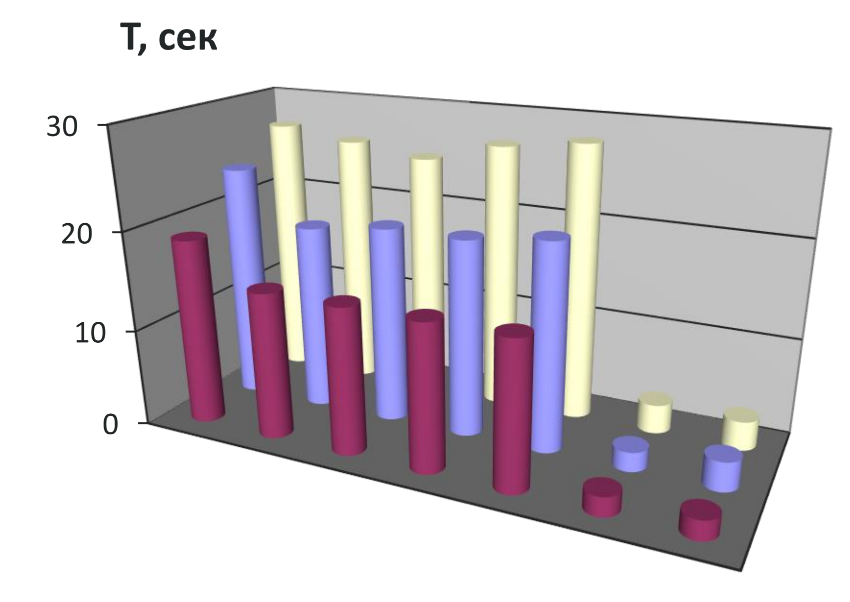
Лечим хронические заболевания





«Хронические заболевания» БД – 1

1. Первое проявление
   в виде проблемы с индексами: при объединении двух основных по размеру таблиц возникал
   Sec Scan вместо обращения по индексу.



Подробности далее

После ~~исправления~~ симптоматического лечения
скорость чтения расписаний **возросла в среднем в 15 раз (временно)**



Проявление проблемы с Sec Scan

GID_Events
Train_EID
…

142 млн.строк

TMP_Trains
Train_EID
…

3682 строки

```
SELECT E.*
FROM   GID_Events E INNER JOIN
       TMP_Trains T
       ON(T.Train_EID=E.Train_EID);
```

GID_Events имеет индекс по Train_EID

QUERY PLAN

```
Hash Semi Join  (cost=170.84..4792890.47 rows=11787245 width=286) (actual time=13254.652..135489.742 rows=816779 loops=1)
  Hash Cond: (e.train_eid = t.train_eid)
  ->  Seq Scan on gid_events e  (cost=0.00..4287162.24 rows=142637824 width=286) (actual time=0.217..117560.619 rows=136429750 loops=1)
  ->  Hash  (cost=124.82..124.82 rows=3682 width=8) (actual time=2.882..2.892 rows=3682 loops=1)
        Buckets: 4096  Batches: 1  Memory Usage: 176kB
        ->  Seq Scan on tmp_trains t  (cost=0.00..124.82 rows=3682 width=8) (actual time=0.036..2.109 rows=3682 loops=1)
Planning Time: 5.428 ms
Execution Time: 135523.087 ms
```



Паллиативное решение проблемы с Sec Scan

Увеличить глубину сбора статистики (см. параметр БД default_statistics_target; 1<=>300 страниц) по колонке с ПК:

```
ALTER TABLE GID_Events ALTER COLUMN Train_EID SET STATISTICS 10000;

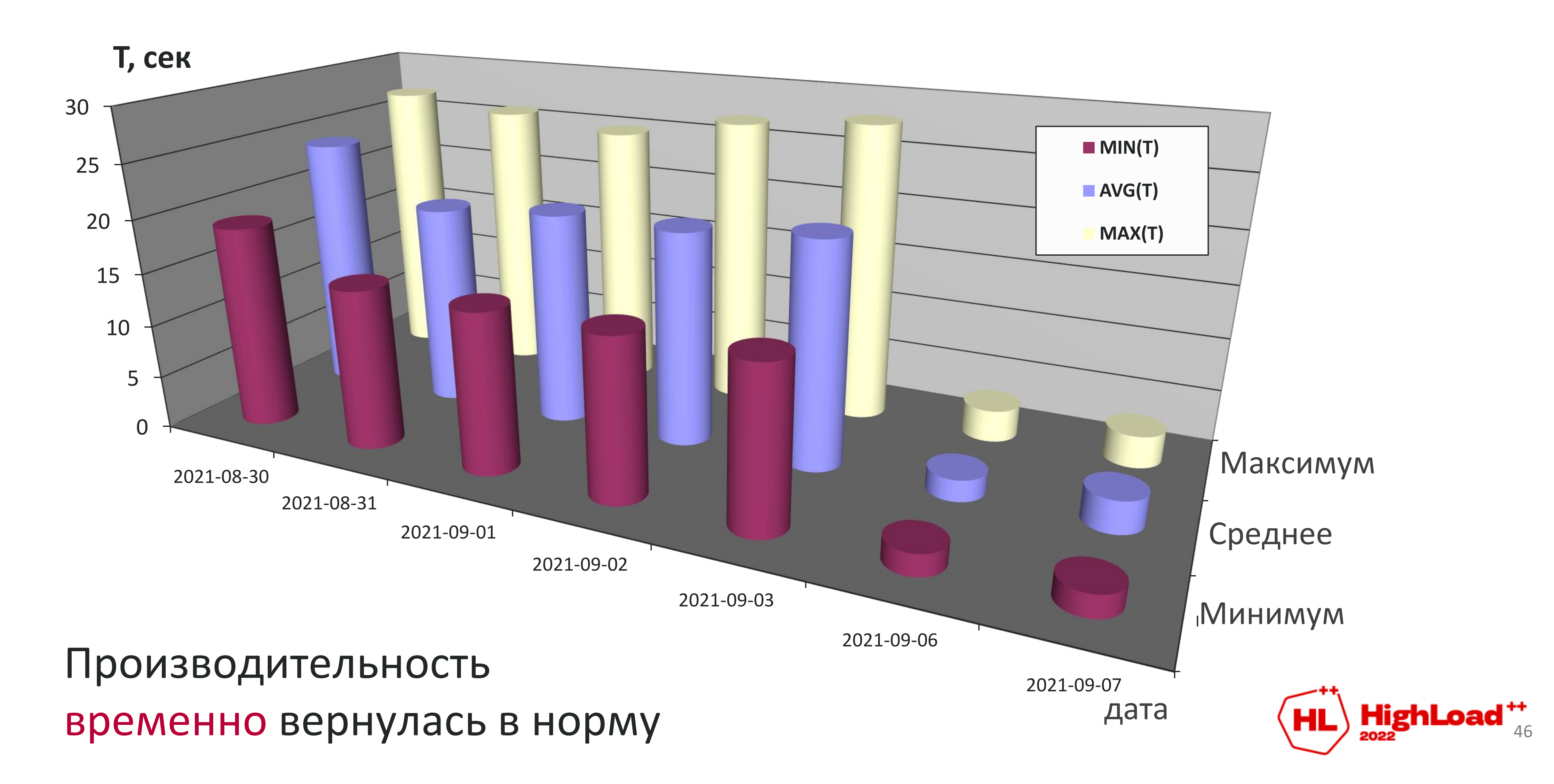
ANALYZE GID_Events;
```

```
QUERY PLAN
Nested Loop  (cost=134.59..774248.40 rows=193491 width=286) (actual time=30.496..6586.509 rows=816779 loops=1)
  ->  HashAggregate  (cost=134.03..170.84 rows=3682 width=8) (actual time=3.238..7.687 rows=3682 loops=1)
        Group Key: t.train_eid
        ->  Seq Scan on tmp_trains t (cost=0.00..124.82 rows=3682 width=8) (actual time=0.034..1.865 rows=3682 loops=1)
  ->  Index Scan using gid_events_pk on gid_events e  (cost=0.57..209.70 rows=53 width=286) (actual time=0.762..1.674 rows=222 loops=3682)
        Index Cond: (train_eid = t.train_eid)
Planning Time: 6.667 ms
Execution Time: 6619.910 ms
```

«Ускорение» в 20 раз!



Время выполнения getTrainsData



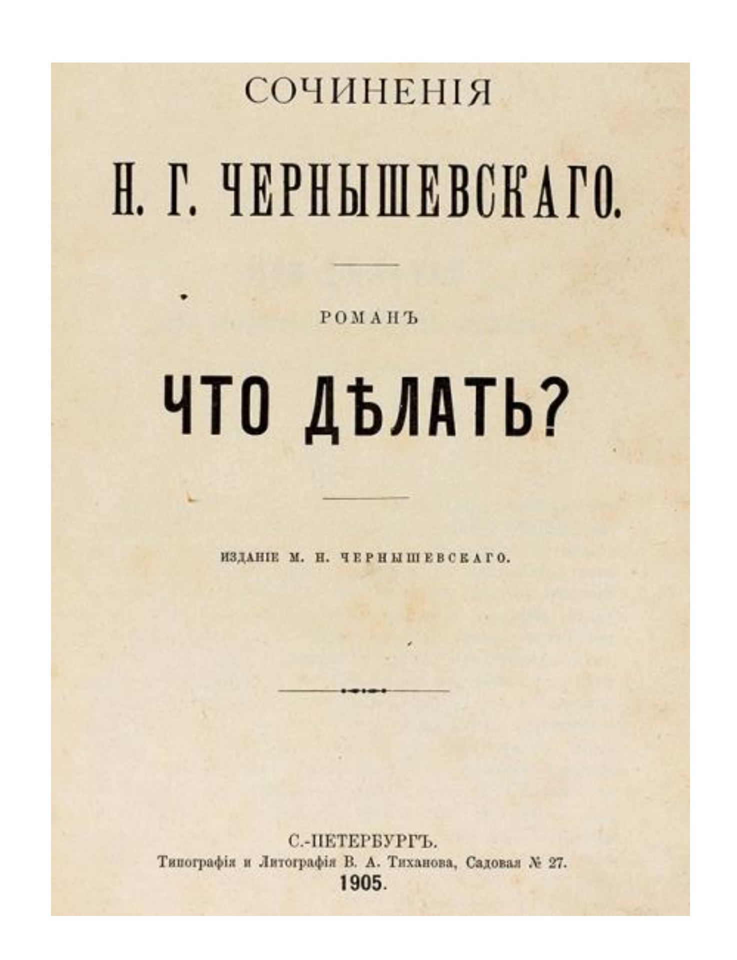
Производительность
временно вернулась в норму

«Хронические заболевания» БД – 2

2. Со временем скорость работы снизилась до неприемлемых значений, при объединении двух основных по размеру таблиц снова возникал
   Sec Scan вместо обращения по индексу.

   Прежний метод не помогал!

   Перестроение индексов
   не помогало тоже!


СОЧИНЕНІЯ

Н. Г. ЧЕРНЫШЕВСКАГО.

РОМАНЪ

ЧТО ДѢЛАТЬ?

ИЗДАНІЕ М. Н. ЧЕРНЫШЕВСКАГО.

С.-ПЕТЕРБУРГЪ.
Типографія и Литографія В. А. Тиханова, Садовая № 27.
1905.



Выход найден!



В Oracle были хинты – поищем их в PostgreSQL
и заставим использовать индексы!



«Хинты» в PostgreSQL (нет)

- Хинтов в стиле Oracle в PostgreSQL нет (есть **pg-hint-plan** в Pro), но давать советы планировщику все же можно
- Менять нужно только настройки текущей сессии и не забыть восстанавливать исходное состояние настроечного параметра

```
SET LOCAL enable_seqscan = OFF; -- Порицаем использование SecScan

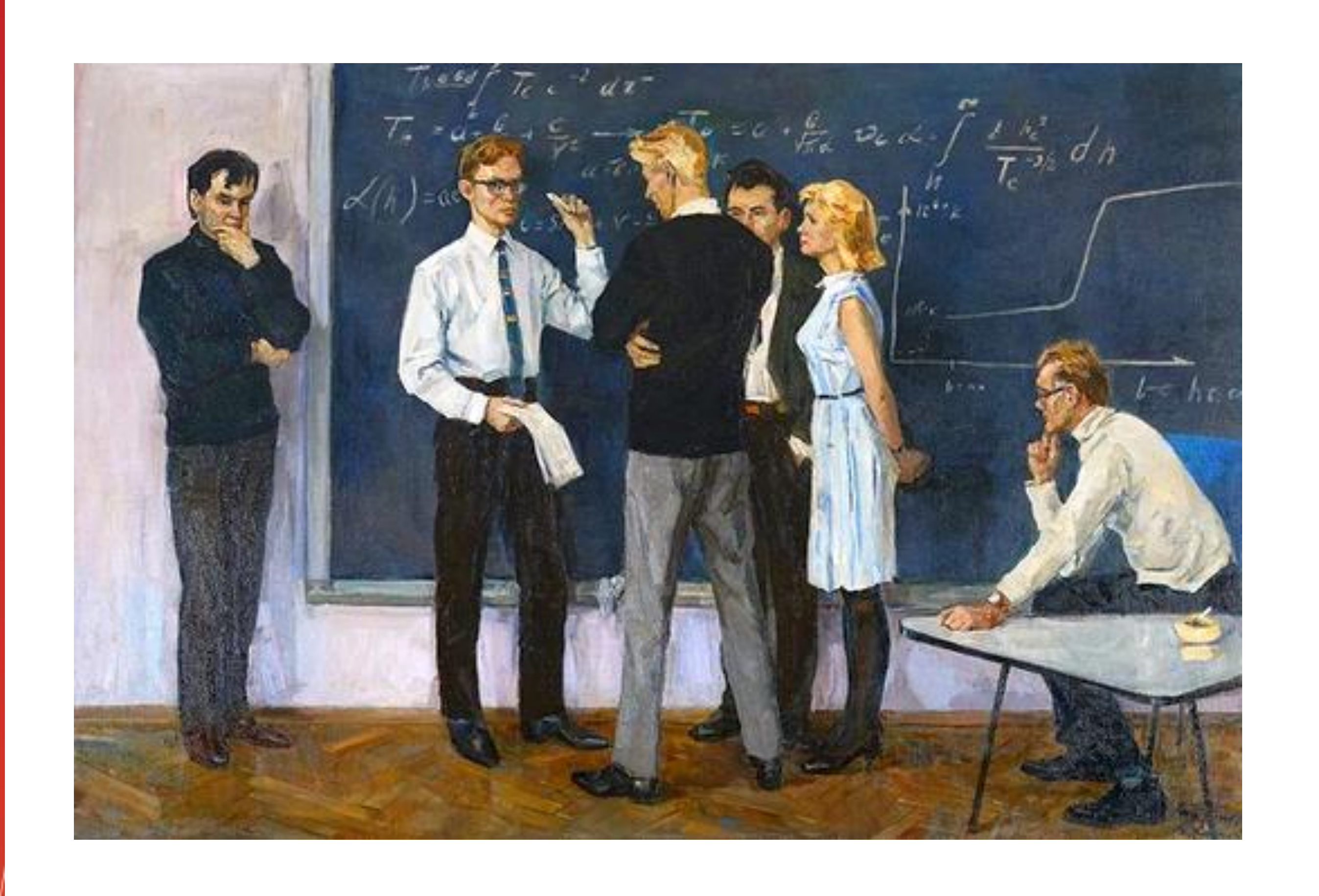
… -- Выполняем проблемный SQL-запрос

SET LOCAL enable_seqscan TO DEFAULT; -- Возвращаем как было
```

Ненадолго помогает, но так как причина не выявлена и не устранена, скоро снова становится плохо…



Причина вычислена!



Настоящая причина SecScan – **autovacuum**

Настройки **autovacuum** по умолчанию
не отвечают требованиям нашей БД
и приводят к такому результату!

Потребовалось изучить **VACUUM** и **autovacuum**,
а также внести изменения в настройки по умолчанию!

Спасибо Егору Рогову и Алексею Лесовскому,
их статьи по теме очень помогли!

https://habr.com/ru/company/postgrespro/blog/452762/
https://dataegret.com/category/autovacuum/



Корень проблемы: распухание таблиц (bloating)

- Операторы DELETE и UPDATE приводят к **появлению устаревших версий строк**, которые занимают место в таблице/индексе, но не могут быть повторно использованы до завершения процедуры **autovacuum** или **VACUUM**
- Если настройки **autovacuum** таковы, что он не успевает отработать интенсивно обновляемые таблицы, то размер таблиц и индексов увеличивается, падает эффективность кэшей, не собирается статистика и портятся планы выполнения – перестают использоваться индексы и возникает Sec Scan
- Необходимо своевременно обнаруживать, а еще лучше – предотвращать распухание таблиц

Методы борьбы с распуханием таблиц

Методы	+	−
autovacuum	• Работает автоматически • Процесс внутри СУБД	• Не всегда успевает при настройках по умолчанию
VACUUM	• Не блокирует • Работает быстро • Распараллеливается	• Не уменьшает размер БД (почти) • Не работает из ХП
VACUUM FULL	• Уменьшает размер БД	• Блокирует даже чтение • Не работает из ХП
pg_repack	• Уменьшает размер БД • Блокирует ненадолго	• Нештатное расширение • Устанавливает триггеры

Не забываем делать ANALYZE после реорганизации таблиц!



Симптомы проблем с autovacuum

- Простой SELECT из интерфейсной UNLOGGED-таблицы внезапно стал занимать 95% времени работы процедуры независимо от числа записей в этой таблице, вплоть до 0
- По состоянию UNLOGGED-таблиц мы видим, что `autovacuum` к ним давно не применялся!

table_name	n_live_tuples	n_dead_tuples
tmp_trains	0	521 457
tmp_timetablepoints	0	15 048 410
tmp_traincalendars	0	8 544 236



Зачем настраивать autovacuum?

- До 60% содержимого основных таблиц в нашей БД обновляется за неделю
- Настройки **autovacuum** по умолчанию таковы, что отработать интенсивно обновляемые таблицы он не успевает. Кстати, начиная с версии 12 настройки по умолчанию более активные
- Статистика по таблице и индексу собирается в ходе выполнения **autovacuum**; если он не работает, то статистика не собирается и устаревает; поэтому перестают использоваться индексы –> Sec Scan
- Параметры работы **autovacuum** можно настроить индивидуально для отдельных таблиц



Настройка autovacuum

- Устанавливаем индивидуальные значения для часто обновляемых таблиц командой ALTER TABLE trains SET(…)

Параметр	Значение	Назначение параметра
autovacuum_vacuum_scale_factor	0	Пороговая доля строк в таблице для принятия решения о начале вакуумирования
autovacuum_vacuum_threshold	10000	Пороговое число «мертвых» строк в таблице для принятия решения о начале вакуумирования
autovacuum_vacuum_cost_delay	5	Перерыв между циклами автовакуума, мс (вместо 20 мс по умолчанию в 11-й версии)
autovacuum_vacuum_cost_limit	1000	Стоимость одного цикла автовакуума (вместо 200 по умолчанию)

Помогает надолго, но не навсегда…



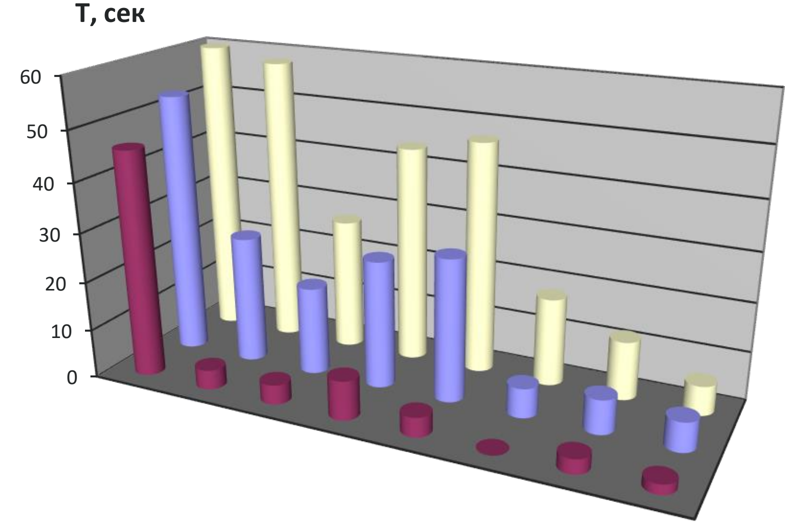
Практика борьбы с распуханием таблиц: autovacuum + VACUUM + pg_repack

- 1 раз в месяц минимизируем основные таблицы и индексы БД с помощью **pg_repack** в плановое окно техобслуживания

- Для профилактики распухания ежедневно выполняем **VACUUM** в периоды наименьшей нагрузки (ночью)

- Индивидуально настроен **autovacuum**

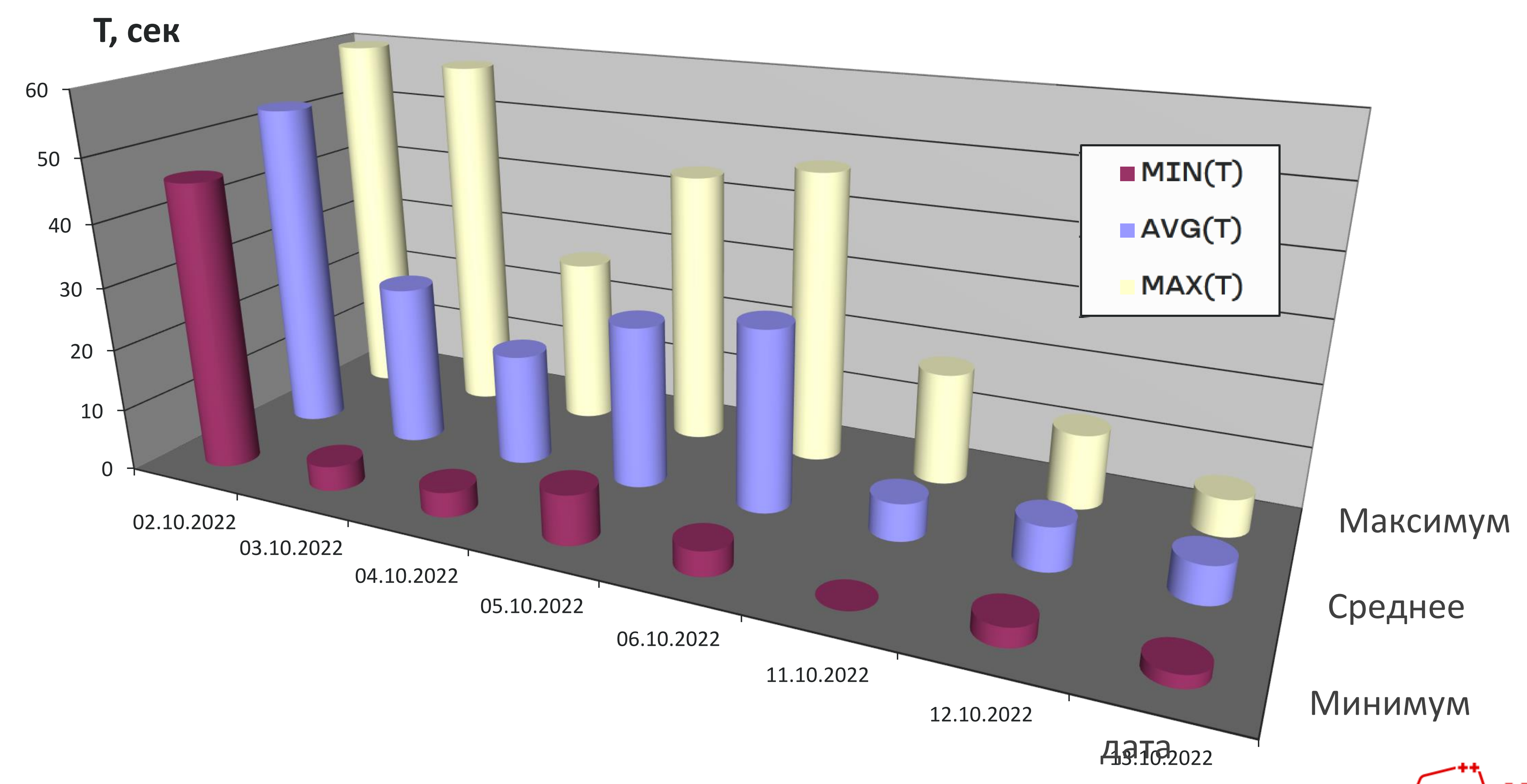
- Результат: процедура getTrainsData в среднем **ускорилась в 3 раза!**



Подробности далее



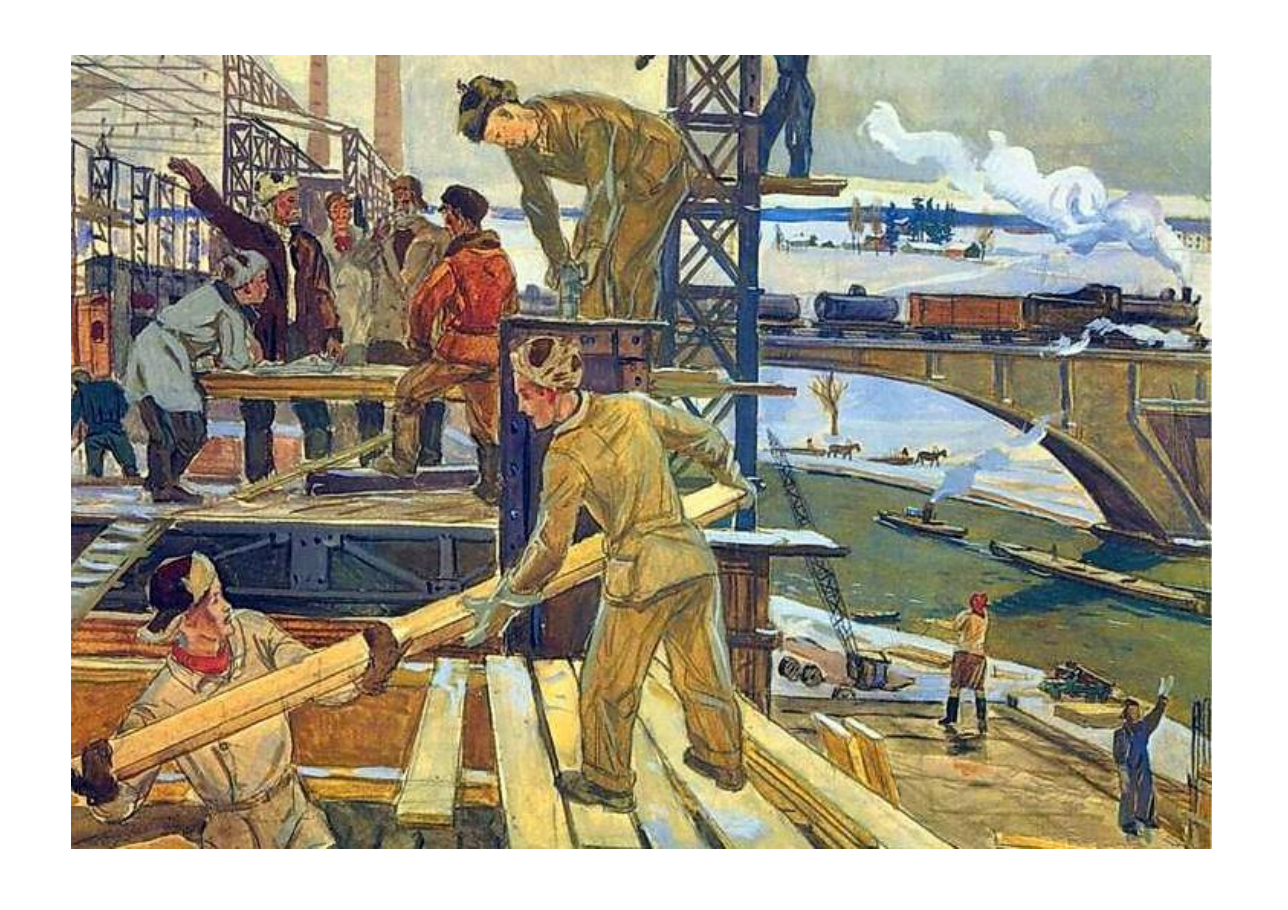
Результаты борьбы с распуханием таблиц



Процедуры getTrainsData ускорилась в 3 раза

HL++ HighLoad++ 2022

Строим новую систему



Новая БД: проектируем ЭЛЬБРУС–М «с нуля»



Макромодель ЭЛЬБРУС–М для прогноза продвижения поездопотоков и оценки инфраструктурных и управляющих решений



Можно повлиять
на производительность системы
на двух недоступных ранее уровнях!

- Выбор структуры данных и архитектуры приложения
- Разработка API и выбор способа хранения данных



Выбор структуры данных и архитектуры

- Выполнена глубокая нормализация данных о расписаниях

ОБЪЕКТЫ РАСПИСАНИЙ

1) Поезда
2) Точки ниток поездов
3) Календари

Нормализация →

ОБЪЕКТЫ РАСПИСАНИЙ

1) Нитки поездов
2) Точки ниток поездов
3) Поезда
4) Изменения поездов по ходу движения
5) Календари
6) Маршруты
7) Перегоны маршрутов

Объем данных одного расписания уменьшился в 10-100 раз



Выбор API и способа хранения данных

- Активное использование партицирования
- Шаблоны для создания временных таблиц
- Предварительный расчет и хранение промежуточных данных для аналитических запросов
- Отказ от использования UNLOGGED-таблиц в пользу традиционных временных таблиц PostgreSQL – это выигрыш до 40% в производительности на тяжелых ХП.
  Для этого перед вызовом ХП вызываем функцию `prepareCall('`*ИМЯ_ПРОЦЕДУРЫ*`')` для создания необходимых для работы API временных таблиц в данной сессии



Продолжение следует!

- **Импортозамещение состоялось и приносит реальную пользу**
- **Новое приложение и новый тип БД – новые вызовы**
- **Впереди – Postgres Pro**
- **Скучать не придется!**





Анатолий
Анфиногенов,

ВНИИЖТ

anfinogenov.anatoly@vniizht.ru

Обратная связь
и комментарии по докладу
– по ссылке:



Понравился доклад–
голосуйте по ссылке:

